Outline

The web from a security perspective
SQL injection
Announcements intermission
Cross-site scripting
More cross-site risks

Once upon a time: the static web

HTTP: stateless file download protocol
- TCP, usually using port 80
HTML: markup language for text with formatting and links
- All pages public, so no need for authentication or encryption

Web applications

The modern web depends heavily on active software
- Static pages have ads, paywalls, or "Edit" buttons
- Many web sites are primarily forms or storefronts
- Web hosted versions of desktop apps like word processing

Server programs

Could be anything that outputs HTML
- In practice, heavy use of databases and frameworks
- Wide variety of commercial, open-source, and custom-written
- Flexible scripting languages for ease of development
  - PHP, Ruby, Perl, etc.

Client-side programming

Java: nice language, mostly moved to other uses
ActiveX: Windows-only binaries, no sandboxing
Glad to see it on the way out
Flash and Silverlight: last important use was DRM-ed video
Core language: JavaScript

JavaScript and the DOM

JavaScript (JS) is a dynamically-typed prototype-OO language
- No real similarity with Java
Document Object Model (DOM): lets JS interact with pages and the browser
- Extensive security checks for untrusted-code model

Same-origin policy

Origin is a tuple (scheme, host, port)
- E.g., (http, www.umn.edu, 80)
Basic JS rule: interaction is allowed only with the same origin
Different sites are (mostly) isolated applications
GET, POST, and cookies

- **GET** request loads a URL, may have parameters delimited with `?`, `&`, `-`
  - Standard: should not have side-effects
- **POST** request originally for forms
  - Can be larger, more hidden, have side-effects
- **Cookie**: small token chosen by server, sent back on subsequent requests to same domain

User and attack models

- "Web attacker" owns their own site
  - (www.attacker.com)
  - And users sometimes visit it
  - Realistic reasons: ads, SEO
- "Network attacker" can view and sniff unencrypted data
  - Unprotected coffee shop WiFi

Outline

- The web from a security perspective
  - SQL injection
  - Announcements intermission
  - Cross-site scripting
  - More cross-site risks

Relational model and SQL

- Relational databases have *tables* with *rows* and *single-typed columns*
  - Used in web sites (and elsewhere) to provide scalable persistent storage
  - Allow complex *queries* in a declarative language SQL

Example SQL queries

- SELECT name, grade FROM Students WHERE grade < 60 ORDER BY name;
- UPDATE Votes SET count = count + 1 WHERE candidate = 'John';

Template: injection attacks

- Your program interacts with an interpreted language
  - Untrusted data can be passed to the interpreter
  - Attack data can break parsing assumptions and execute arbitrary commands

SQL + injection

- Why is this named most critical web app. risk?
  - Easy mistake to make systematically
  - Can be easy to exploit
  - Database often has high-impact contents
  - Eg, logins or credit cards on commerce site

Strings do not respect syntax

- Key problem: assembling commands as strings
  - "WHERE name = '$name';"
  - Looks like `$name` is a string
  - Try `$name = "me' OR grade > 80; --"
**Using tautologies**

- Tautology: formula that's always true
- Often convenient for attacker to see a whole table
- Classic: \( \text{OR } 1=1 \)

**Non-string interfaces**

- Best fix: avoid constructing queries as strings
- SQL mechanism: prepared statement
  - Original motivation was performance
- Web languages/frameworks often provide other syntax

**Retain functionality: escape**

- *Sanitizing* data is transforming it to prevent an attack
- *Escaped* data is encoded to match language rules for literal
  - E.g., `\"` and `\n` in C
- But many pitfalls for the unwary:
  - Differences in escape syntax between servers
  - Must use right escape for context: not everything's a string

**Lazy sanitization: allow-listing**

- Allow only things you know to be safe/intended
- Error or delete anything else
- Short allow-list is easy and relatively easy to secure
  - E.g., digits only for non-negative integer
- But, tends to break benign functionality

**Poor idea: deny-listing**

- Space of possible attacks is endless, don't try to think of them all
- Want to guess how many more comment formats SQL has?
- Particularly silly: deny 1–1

**Attacking without the program**

- Often web attacks don't get to see the program
  - Not even binary, it's on the server
- Surmountable obstacle:
  - Guess natural names for columns
  - Harvest information from error messages

**Blind SQL injection**

- Attacking with almost no feedback
- Common: only "error" or "no error"
- One bit channel you can make yourself: if \( x \) delay 10 seconds
- Trick to remember: go one character at a time

**Injection beyond SQL**

- Earlier: shell commands, format strings
- XPath/XQuery: queries on XML data
- LDAP: queries used for authentication
- Next up: XSS
Outline
The web from a security perspective
SQL injection
Announcements intermission
Cross-site scripting
More cross-site risks

Note to early readers
This is the section of the slides most likely to change in the final version
If class has already happened, make sure you have the latest slides for announcements

Outline
The web from a security perspective
SQL injection
Announcements intermission
Cross-site scripting
More cross-site risks

XSS: HTML/JS injection
Note: CSS is “Cascading Style Sheets”
Another instance of injection template
Attacker supplies HTML containing JavaScript (or occasionally CSS)
OWASP’s most prevalent weakness
A category unto itself
Easy to commit in any dynamic page construction

Why XSS is bad (and named that)
attacker.com can send you evil JS directly
But XSS allows access to bank.com data
Violates same-origin policy
Not all attacks actually involve multiple sites

Reflected XSS
Injected data used immediately in producing a page
Commonly supplied as query/form parameters
Classic attack is link from evil site to victim site

Persistent XSS
Injected data used to produce page later
For instance, might be stored in database
Can be used by one site user to attack another user
E.g., to gain administrator privilege

DOM-based XSS
Injection occurs in client-side page construction
Flaw at least partially in code running on client
Many attacks involve mashups and inter-site communication
No string-free solution
- For server-side XSS, no way to avoid string concatenation
- Web page will be sent as text in the end
- XSS especially hard kind of injection

Danger: complex language embedding
- JS and CSS are complex languages in their own right
- Can appear in various places with HTML
  - But totally different parsing rules
- Example: "..." used for HTML attributes and JS strings
  - What happens when attribute contains JS?

Danger: forgiving parsers
- History: handwritten HTML, browser competition
- Many syntax mistakes given "likely" interpretations
- Handling of incorrect syntax was not standardized

Sanitization: plain text only
- Easiest case: no tags intended, insert at document text level
- Escape HTML special characters with entities like \&lt; for <
- OWASP recommendation: & < > " ' /

Sanitization: context matters
- An OWASP document lists 5 places in a web page you might insert text
  - For the rest, "don't do that"
- Each one needs a very different kind of escaping

Sanitization: tag allow-listing
- In some applications, want to allow benign markup like <b>
- But, even benign tags can have JS attributes
- Handling well essentially requires an HTML parser
  - But with an adversarial-oriented design

Don't deny-list
- Browser capabilities continue to evolve
- Attempts to list all bad constructs inevitably incomplete
- Even worse for XSS than other injection attacks

Filter failure: one-pass delete
- Simple idea: remove all occurrences of <script>
- What happens to <scr<script>ipt>?
Filter failure: UTF-7
- You may have heard of UTF-8
  - Encode Unicode as 8-bit bytes
- UTF-7 is similar but uses only ASCII
- Encoding can be specified in a `<meta>` tag, or some browsers will guess
  +ADw-script+AD4-

Filter failure: event handlers
- `<img onmouseover="alert('xss')">`
  - Put this on something the user will be tempted to click on
  - There are more than 100 handlers like this recognized by various browsers

Use good libraries
- Coding your own defenses will never work
- Take advantage of known good implementations
- Best case: already built into your framework
  - Disappointingly rare

Content Security Policy
- Added HTTP header, W3C recommendation
- Lets site opt-in to stricter treatment of embedded content, such as:
  - No inline JS, only loaded from separate URLs
  - Disable JS `eval` et al.
- Has an interesting violation-reporting mode

Outline
The web from a security perspective
- SQL injection
- Announcements intermission
- Cross-site scripting
- More cross-site risks

HTTP header injection
- Untrusted data included in response headers
- Can include CRLF and new headers, or premature end to headers
- AKA "response splitting"

Content sniffing
- Browsers determine file type from headers, extension, and content-based guessing
  - Latter two for ~1% server errors
- Many sites host "untrusted" images and media
- Inconsistencies in guessing lead to a kind of XSS
  - E.g., "chimera" PNG-HTML document

Cross-site request forgery
- Certain web form on bank.com used to wire money
- Link or script on evil.com loads it with certain parameters
  - Linking is exception to same-origin
- If I'm logged in, money sent automatically
CSRF prevention

- Give site's forms random-nonce tokens
  - Eg, in POST hidden fields
  - Not in a cookie, that's the whole point
- Reject requests without proper token
  - Or, ask user to re-authenticate
- XSS can be used to steal CSRF tokens

Open redirects

- Common for one page to redirect clients to another
- Target should be validated
  - With authentication check if appropriate
- Open redirect: target supplied in parameter with no checks
  - Doesn't directly hurt the hosting site
  - But reputation risk, say if used in phishing
  - We teach users to trust by site